Лабораторная работа №4. Клиент. Сервер.

Теоретическая часть

Потоки

Поток – выполняющийся программный код. Основной поток связан с главной функцией main() программы, рабочие потоки выполняют параллельно другие функции. Например, обработка событий в Java осуществляется рабочими потоками. Потоки реализуются с помощью класса Thread и функционального интерфейса Runnable, при этом переопределяется потоковая функция run().

Два способа создания потоков

Поток является объектом класса Thread. Имеется два способа создания потоков в Java:

1) поток создаётся из класса, наследующего класс Thread и переопределяющего потоковую функцию run() класса Thread. Запуск потока осуществляется функцией start() класса Thread применительно к созданному объекту потока.

2) Создаётся два объекта. Вначале создаётся объект класса, осуществляющего интерфейс Runnable и переопределяющего надлежащим образом потоковую функцию run(). Затем надо создаётся объект потока класса Thread, передав в качестве аргумента конструктора только что созданный объект класса с переопределённой потоковой функцией run(). К полученному объекту потока применить функцию start(). Ниже представлены оба способа.

```
/* Потоки*/
class A extends Thread {// Класс потокового объекта
 public void run() {
  for(int i=0; i < 5; i++) {
   System.out.print ("A");
   try {Thread.sleep (100);}
   catch (InterruptedException e){}
  }
 }
}
class B implements Runnable {// Класс с потоковой функцией
 public void run() {
  for (int i=0; i < 5; i++) {
   System.out.print ("B");
   try {Thread.sleep (100);}
   catch (InterruptedException e){}
  }
 }
}
class TestThread {
 public static void main (String[] args) {
  A a = new A();//Первый способ создания потока
  B b = new B();//Второй способ создания потока (Шаг.1)
  Thread t= new Thread (b,"thread");//Второй способ создания потока(Шаг.2)
  a.start();//Запуск потоков A
  t.start();//Запуск потоков B
 }
```

```
}
/*Result: ABABABABAB*/
```

В примере программы созданы два объекта потока – a и t. В соответствии с правилами с этими потоками связаны одноимённые функции run(), первая из которых выдаёт на консоль строку “A”, а вторая – “B”. В цикле каждой потоковой функции вызывается статическая функция sleep() класса Thread, позволяя потокам параллельно выдать строки на консоль. Java обязывает на уровне компиляции применить для функции sleep() перехват try-catch исключения InterruptedException.

Как уже говорилось, любой класс может наследовать только один базовый класс (суперкласс). Наследование одного класса Thread лишает не только возможности наследования иных классов, но и ограничивает количество созданных потоков в классе только одним.

Использование интерфейса Runnable позволяет создать множество классов, наследующих этот интерфейс и переопределяющих функцию run() интерфейса Runnable. Затем по второму способу можно создать множество потоковых объектов и запустить их различные потоковые функции.

Использование интерфейсной ссылки в конструкторе класса Thread позволило передать в потоковые объекты различные одноимённые потоковые функции run().

Синхронизация потоков

Обрабатывая параллельно данные, потоки могут вызвать тупиковые ситуации, приводящие к блокированию программы. Поэтому применение потоков требует освобождения или синхронизации разделяемых ресурсов, то есть ресурсов, употребляемых одновременно разными потоками.

Пример возникновения конфликта.

Пусть имеются два ресурса – R1 и R2. И пусть имеются два потока T1 и T2. Пусть поток T1 владеет ресурсом R1, а поток T2 – ресурсом R2. И в параллельно выполняющихся потоках возник момент, когда потоку T1 потребовался ресурс R2, а потоку T2 – ресурс R1. Если ни один из потоков не освободит принадлежащий ему ресурс, программа прекратит выполнение – произойдёт приостановка выполнения каждого потока (тупиковая ситуация).

Для разрешения тупиковых ситуаций надлежит строго выполнять правило: любой поток перед применением какого-либо ресурса обязан освободить все уже ненужные ему ресурсы и обязательно освободить все используемые им ресурсы перед завершением выполнения.

Если же одновременное использование некоторого разделяемого ресурса несколькими потоками приводит к неуправляемому изменению и применению данных этого ресурса, то надлежит синхронизировать выполнение этих потоков, дозволяя потокам только отдельно использовать ресурс.

Оператор synchronized, размещённый в некотором потоке, превращает последовательность операторов его блока в критическую секцию, при выполнении которой другие потоки блокируются. Также применяется ключевое слово synchronized к функциям для обеспечения раздельного использования её потоками.

Java.net

Классы, работающие с сетевыми протоколами, располагаются в пакете java.net, и простейшим из них является класс URL. С его помощью можно сконструировать uniform resource locator (URL), который имеет следующий формат:

protocol://host:port/resource

Здесь:
protocol – название протокола, используемого для связи;
host – IP-адрес, или DNS-имя сервера, к которому производится обращение;
port – номер порта сервера (если порт не указан, то используется значение по умолчанию для указанного протокола);
resource – имя запрашиваемого ресурса, причем, оно может быть составным, например:

ftp://myserver.ru/pub/docs/Jav a/JavaCourse.txt

Затем можно воспользоваться функцией openStream(), которая возвращает InputStream, что позволяет считать содержимое ресурса. Например, следующая программа при помощи LineNumberReader() считывает первую страницу сайта http://www.ru и выводит ее на консоль:

```
import java.io.*;
import java.net.*;
public class Net {
 public static void main(String args[]) {
  try {
   URL url = new URL("http://www.ru");
   LineNumberReader r = new LineNumberReader(
   new InputStreamReader(url.openStream()));
   String s = r.readLine();
   while (s!=null) {
    System.out.println(s);
    s = r.readLine();
   }
   System.out.println(r.getLineNumber());
   r.close();
  }
  catch (MalformedURLException e) {e.printStackTrace();}
  catch (IOException e) {e.printStackTrace();}
 }
}
```

Из программы видим, что работа с сетью, как и работа с потоками, требует дополнительной работы с исключительными ситуациями. Ошибка MalformedURLException появляется в случае, если строка с URL содержит ошибки.

Более функциональным классом является URLConnection, который можно получить с помощью функции класса URL.openConnection(). У этого класса есть две функции – getInputStream() (именно с его помощью работает URL.openStream()) и getOutputStream(), которую можно использов ать для передачи данных на сервер, если он поддерживает такую операцию (многие публичные web-серверы закрыты для таких действий).

Класс URLConnection является абстрактным. Виртуальная машина предоставляет реализации этого класса для каждого протокола, например, в том же пакете java.net определен класс HttpURLConnection. Понятно, что классы URL и URLConnection предоставляют возможность работы через сеть на прикладном уровне с помощью высокоуровневых протоколов.

Пакет java.net также предоставляет доступ к протоколам более низкого уровня – TCP и UDP. Для этого сначала надо ознакомиться с классом InetAddress, который является Internet-адресом, или IP. Экземпляры этого класса создаются не с помощью конструкторов, а с помощью статических функций:

```
InetAddress getLocalHost()
InetAddress getByName(String name)
InetAddress[] getAllByName(String name)
```

Первая функция возвращает IP-адрес машины, на которой исполняется Java-программа. Вторая функция возвращает адрес сервера, чье имя передается в качестве параметра. Это может быть как DNS-имя, так и числовой IP, записанный в виде текста, например, "67.11.12.101".

Третья функция определяет все IP-адреса указанного сервера.

TCP

Для работы с TCP-протоколом используются классы Socket и ServerSocket. Первым создается ServerSocket – сокет на стороне сервера. Его простейший конструктор имеет только один параметр – номер порта, на котором будут приниматься входящие запросы. После создания вызывается функция accept(), которая приостанавливает выполнение программы и ожидает, пока какой-нибудь клиент не инициирует соединение. В этом случае работа сервера возобновляется, а функция возвращает экземпляр класса Socket для взаимодействия с клиентом:

```
try {
 ServerSocket ss = new ServerSocket(3456);
 Socket client=ss.accept();
 // Метод не возвращает
 // управление, пока не подключится клиент
}catch (IOException e) { e.printStackTrace();}
```

Клиент для подключени я к серверу также использует класс Socket. Его простейший конструктор принимает два параметра - адрес сервера (в виде с трок и, или экземпляра InetAddress) и номер порта. Если сервер принял запрос, то сокет конструируется успешно и далее можно воспользоваться функциями getInputStream() или getOutputStream(). Пример работы с конструктором класса Socket:

```
try {
 Socket s = new Socket("localhost", 3456);
 InputStream is = s.getInputStream();
 is.read();
} catch (UnknownHostException e) {e.printStackTrace();
} catch (IOException e) {e.printStackTrace();}
```

Обратите внимание на обработку исключительной ситуации UnknownHostException, которая будет генерироваться, если виртуальная машина с помощью операционной

системы не сможет распознать указанный адрес сервера в случае, если он задан строкой. Если же он задан экземпляром InetAddress, то эту ошибку надо обрабатывать при вызове статических методов данного класса.

На стороне сервера класс Socket используетс я точно таким же образом – через методы getInputStream() и getOutputStream(). Обратите внимание, что при завершении некоторго блока кода либо приложения следует вызывать функцию close() для открытых сокетов.

Класс ServerSocket имеет конструктор, в который передается, кроме номера порта, еще и адрес машины. Это должен быть именно локальный адрес машины, иначе возникнет ошибка. Аналогично, класс Socket имеет расширенный конструктор для указания как локального адреса, с которого будет устанавливаться соединение, так и локального порта (иначе операционная система выделяет произвольный свободный порт).

Можно воспользоваться функцией setSoTimeout(int timeout) класса ServerSocket, чтобы указать время в миллисекундах, на протяжении которого нужно ожидать подключение клиента. Это позволяет серверу не "зависать", если никто не пытается начать с ним работать. Тайм-аут задается в миллисекундах, нулевое значение означает бесконечное время ожидания.

Важно подчеркнуть, что после установления соединения с клиентом сервер выходит из функции accept(), то есть перестает быть готовым принимать новые запросы. Однако, как правило, желательно, чтобы сервер мог работать с несколькими клиентами одновременно. Для этого необходимо при подключении очередного пользователя создавать новый поток исполнения, который будет обслуживать его, а основной поток снова войдет в функцию accept().

Пример. Простой TCP сервер

```
import java.io.*;
import java.net.*;
public class TCPServer {
 public static final int PORT = 2500;
 private static final int TIME_SEND_SLEEP = 100;
 private static final int COUNT_TO_SEND = 10;
 private ServerSocket servSocket;
 public static void main(String[] args) {
  TCPServer tcpServer = new TCPServer();
  tcpServer.go();
 }
 public TCPServer(){
  try{
   servSocket = new ServerSocket(PORT);
  }catch(IOException e){
   System.err.println("Не удаётся открыть сокет для сервера: " + e.toString());
  }
 }
 public void go(){
  class Listener implements Runnable{
   Socket socket;
   public Listener(Socket aSocket){
    socket = aSocket;
   }
 public void run(){
  try{
```

```
   System.out.println("Слушатель запущен");
   int count = 0;
   OutputStream out = socket.getOutputStream();
   OutputStreamWriter writer = new OutputStreamWriter(out);
   PrintWriter pWriter = new PrintWriter(writer);
   while(count < COUNT_TO_SEND){
    count++;
    pWriter.print(((count>1) ? "," : "") + "говорит " + count);
    Thread.sleep(TIME_SEND_SLEEP);
   }
   pWriter.close();
   }catch(IOException e){
   System.err.println("Исключение: " + e.toString());
   } catch (InterruptedException e) {
    System.err.println("Исключение: " + e.toString());
   }
  }
 }
 System.out.println("Сервер запущен...");
 while(true){
  try{
     Socket socket = servSocket.accept();
     Listener listener = new Listener(socket);
     Thread thread = new Thread(listener);
     thread.start();
    }catch(IOException e){
    System.err.println("Исключение: " + e.toString());
   }
  }
 }
}
```

Пример. Простой TCP клиент

Теперь объявим клиента. Эта программа будет запускать несколько потоков, каждый из которых независимо подключается к серверу, считывает его ответ и выводит на консоль.

```
import java.io.*;
import java.net.*;
public class TCPClient implements Runnable {
 public static final int PORT = 2500;
 public static final String HOST = "localhost";
 public static final int CLIENT_COUNT = 6;
 public static final int READ_BUFFER_SIZE = 10;
 private String name = null;
 public TCPClient(String s){
  name = s;
 }
 public void run(){
 char[] readed = new char[READ_BUFFER_SIZE];
 StringBuffer strBuff = new StringBuffer();
 try{
```

```
   Socket socket = new Socket(HOST, PORT);
   InputStream in = socket.getInputStream();
   InputStreamReader reader = new InputStreamReader(in);
   while(true){
    int count = reader.read(readed, 0, READ_BUFFER_SIZE);
    if(count == -1) break;
    strBuff.append(readed, 0, count);
    Thread.yield();
   }
  } catch (UnknownHostException e) {
   System.err.println("Исключение: " + e.toString());
  } catch (IOException e) {
   System.err.println("Исключение: " + e.toString());
  }
  System.out.println("Клиент " + name + " прочёл: " + strBuff.toString());
 }
 public static void main(String[] args) {
  String name = "имя";
  for(int i = 1; i <= CLIENT_COUNT; i++){
   TCPClient ja = new TCPClient(name+i);
   Thread th = new Thread(ja);
   th.start();
  }
 }
}
```

UDP

Теперь рассмотрим UDP. Для работы с этим протоколом и на стороне клиента, и на стороне сервера используется класс DatagramSocket. У него есть следующие конструкторы:

```
DatagramSocket()
DatagramSocket(int port)
DatagramSocket(int port, InetAddress laddr)
```

При вызове первого конструктора сокет открывается на произвольном доступном порту, что уместно для клиента. Конструктор с одним параметром, задающим порт, как правило, применяется на серверах, чтобы клиенты знали, на каком порту им нужно пытаться устанавливать соединение. Наконец, последний конструкт ор необходим для машин, у которых присутствует несколько сетевых интерфейсов.

После открытия сокетов начинается обмен датаграммами. Они представляются экземплярами класса DatagramPacket. При отсылке сообщения применяется следующий конструктор:

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

Массив содержит данные для отправки (созданный пакет будет иметь длину, равную length), а адрес и порт указывают получателя пакета. После этого вызывается функция send() класса DatagramSocket.

Для получения датаграммы также создается экземпляр класса DatagramPacket, но в конструктор передается лишь массив, в который будут записаны полученные данные

(также указывается ожидаемая длина пакета). Сокет необходимо создать с указанием порта, иначе, скорее всего, сообщение просто не дойдет до адресата. Используется функцию receive() класса DatagramSocket (аналогично функции ServerSocket.accept(), эта функция также не прерывает выполнение потока, пока не придет запрос от клиента).

Если запустить сначала получателя, а затем отправителя, то можно увидеть, что первый напечатает содержимое полученной датаграммы, а потом программы завершат свою работу.

Пример. Простой UDP сервер

```
import java.io.*;
import java.net.*;
public class UDPServer {
 public static final int LENGTH_PACKET = 30;
 public static final String HOST = "localhost";
 public static final int PORT = 2345;
 public static final byte[] answer = (" приписка от сервера").getBytes();
 public static void main(String[] args) {
  DatagramSocket servSocket = null;
  DatagramPacket datagram;
  InetAddress clientAddr;
  int clientPort;
  byte[] data;
  try{
  servSocket = new DatagramSocket(PORT);
  }catch(SocketException e){
   System.err.println("Не удаётся открыть сокет : " + e.toString());
  }
  while(true){
   try{
// ---------------------------------------------------------
//очень... долгий цикл
//приём данных от клиента
// ---------------------------------------------------------
    data = new byte[LENGTH_PACKET];
    datagram = new DatagramPacket(data, data.length);
    servSocket.receive(datagram);
    System.out.println("Принято от клиента: " + (new String(datagram.getData())).trim());
// ---------------------------------------------------------
//отправка данных клиенту
//адрес и порт можно вычислить из предыдущей сессии приёма, через
//объект класс DatagramPacket - datagram
// ---------------------------------------------------------
    clientAddr = datagram.getAddress();
    clientPort = datagram.getPort();
// ---------------------------------------------------------
//приписывание к полученному сообщению текста " приписка от сервера" и отправка
//результата обратно клиенту
// ---------------------------------------------------------
    datagram.setData(((new String(datagram.getData())).trim()).getBytes());
    data = getSendData(datagram.getData());
    datagram = new DatagramPacket(data, data.length, clientAddr, clientPort);
```

```
    servSocket.send(datagram);
   }catch(IOException e){
    System.err.println("io исключение : " + e.toString());
   }
  }
 }
// ------------------------------------------------------------
//копирование, работа с массивом типа byte
 protected static byte[] getSendData(byte[] b){
 byte[] result = new byte[b.length + answer.length];
 System.arraycopy(b, 0, result, 0, b.length);
 System.arraycopy(answer, 0, result, b.length, answer.length);
 return result;
 }
}
```

Пример. Простой UDP клиент

```
import java.io.*;
import java.net.*;
public class UDPClient {
 public static final int LENGTH_PACKET = 60;
 public static final String HOST = "localhost";
 public static final int PORT = 2345;
 public static void main(String[] args) {
 try{
//-------------------------------------------------------------------
//отправка сообщения на сервер
//-------------------------------------------------------------------
 byte data[] = ("hello!!! % 1234 5-9*6;").getBytes();
 InetAddress addr = InetAddress.getByName(HOST);
 DatagramSocket socket = new DatagramSocket();
 DatagramPacket packet = new DatagramPacket(data, data.length, addr, PORT);
 socket.send(packet);
 System.out.println("Сообщение отправлено...");
//-------------------------------------------------------------------
//приём сообщения с сервера
//-------------------------------------------------------------------
 byte data2[];
 data2 = new byte[LENGTH_PACKET];
 packet = new DatagramPacket(data2, data2.length);
 socket.receive(packet);
 System.out.println((new String(packet.getData())).trim());
//-------------------------------------------------------------------
//закрытие сокета
//-------------------------------------------------------------------
 socket.close();
 }catch(SocketException e){e.printStackTrace();
 }catch(IOException e){e.printStackTrace();}
 }
}
```

Практическая часть

1. Разбиться на пары, чтобы выполнить лабораторную работу на двух компьютерах.
2. Реализовать приложения клиент и сервер (0 – TCP протокол, 1 – UDP протокол).
3. Реализовать в клиенте указание адреса и порта сервера, так: 1 – с консоли ввода приложения, 2 – из командной строки, 3 – из файла настроек[1].
4. Реализовать указание порта для сервера, так: 1 – с консоли ввода приложения, 2 – из командной строки, 3 – из файла настроек.
5. Сообщения, получаемые клиентом с сервера должны записываться в файл «Журнала клиента» путь к которому определяется следующим образом: 1 – с консоли ввода приложения, 2 – из командной строки, 3 – из файла настроек.
6. Сообщения, получаемые сервером от клиента должны записываться в файл «Журнала сервера» путь к которому определяется следующим образом: 1 – с консоли ввода приложения, 2 – из командной строки, 3 – из файла настроек.

1 Если требуется использовать файл настроек, то его (файл настроек) нужно разместить в корневом каталоге проекта либо жёстко указать к нему путь в коде разрабатываемого приложения.

Варианты заданий

Выполнить задание в паре см. в таблицу 2

Таблица №1

№	Задачи сервера
1.	Сервер возвращает клиенту результат выражения (допустимые операции «+», «-», «*», «/»). Операнды передаются одной строкой (например «9+6*5-7.5/8»). В случае не возможности разобрать сервером полученную строку или при переполнении, возникшем при вычислении полученного выражения, сервер присылает клиенту соответствующее уведомление.
2.	Сервер возвращает клиенту результат выражения (допустимые операции «+», «-»). Операнды и операции передаются за раз по одному (например, выражение «3.4+1.6-5=» нужно передавать с помощью трёх сообщений: «3.4+», «1.6-» и «5=», где «=» - признак конца выражения). В случае не возможности разобрать сервером полученную строку или при переполнении, возникшем при вычислении полученного выражения, сервер присылает клиенту соответствующее уведомление.
3.	На сервере есть три двумерных массива данных: целочисленных, вещественных и строковых. Клиент по указанию номера массива данных и ячейки в нём должен быть способен: считывать и перезаписывать её. Если клиент указывает несколько ячеек (возможно в разных массивах данных), то они должны принимать некоторое предустановленное значение. Также, по запросу клиента сервер должен возвращать размерность указанного массива данных. Изменения в массивах должны дублироваться выводом массивов на консоль.
4.	На сервере есть три двумерных массива данных: целочисленных, вещественных и строковых. Клиент по указанию номера массива данных и ячейки в нём должен быть способен: считывать и перезаписывать её. Изменения в массивах должны дублироваться выводом массивов на консоль. На сервере предусмотреть возможность указания «защищённых от изменения»[2] областей массивов данных (запрет на их изменение клиентом). Задать эти «защищённые от изменения» области можно, например, через диапазон индексов или через перечисление индексов указанных массивов данных. Клиент при попытке изменить данные из «защищённых от изменения» областей должен получать от сервера уведомление о том, что у него нет права на такую операцию.
5.	Сервер возвращает клиенту результат решения задачи, полученной как задание в первой лабораторной работе. Необходимо при этом исходные данные для решения задачи передавать с клиента за две или более итераций. То есть запрещается передать исходные данные на сервер за один запрос.

Таблица №2

№	(П.1) TCP / UDP	(П.2) Адрес и порт сервера	(П.3) Порт сервера	(П.4) Журнал клиента	(П.5) Журнал сервера	Задание из табл.№1
1.	0	3	2	1	1	1
2.	0	2	3	2	2	2
3.	1	2	1	3	2	3
4.	0	3	1	2	3	4
5.	1	1	2	3	1	5
6.	0	1	1	1	2	1
7.	1	2	2	2	3	2

2«Защищённые от изменения» области задавать через параметры командной строки.

8.	0	1	3	1	1	3
9.	0	2	1	1	2	4
10.	0	3	2	3	1	5
11.	1	1	1	1	2	1
12.	0	2	3	2	1	2
13.	1	1	1	1	2	3
14.	0	3	2	2	3	4
15.	0	1	2	3	2	5
16.	0	2	3	1	3	1
17.	0	2	1	2	1	2
18.	1	3	2	1	2	3
19.	0	2	3	2	1	4
20.	1	3	1	3	1	5
21.	1	1	2	1	2	1
22.	1	2	1	2	3	2
23.	0	2	2	2	2	3
24.	0	3	1	3	3	4
25.	1	1	2	2	1	5
26.	0	2	3	3	2	1
27.	1	2	2	1	1	2
28.	0	3	3	2	1	3
29.	1	2	1	1	3	4
30.	1	3	2	2	1	5
31.	0	1	1	3	2	1
32.	1	2	2	1	1	2
33.	0	1	2	1	1	3
34.	1	1	3	1	1	4
35.	1	2	1	2	2	5
36.	0	3	2	3	3	1
37.	1	1	2	3	1	2
38.	1	2	3	1	2	3
39.	0	1	2	1	3	4
40.	1	2	3	2	1	5
41.	1	1	1	2	1	1
42.	1	1	1	3	1	2
43.	0	2	2	1	2	3
44.	1	3	3	1	3	4
45.	1	1	1	2	3	5
46.	1	3	2	3	1	3
47.	0	1	1	1	1	4
48.	1	2	1	3	2	5